CS477 - Formal Software Development Methods

Inductive Theorem Proving

Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign

Acknowledgements for Slides and ITP

I would like to warmly thank Prof. José Meseguer for providing the slides used in this topic, and to Manuel Clavel for developing ITP, an inductive theorem prover that we will use in this class.

ITP is implemented fully in Maude and will allow us to use a computer to mechanize inductive proofs.

Mechanizing proofs is an important, if not the most important, aspect of formal methods. Why do we want to mechanize proofs in a computer-assisted manner? The asnwer is simple: because the computer does not let us make mistakes. If proof mistakes are common and acceptable in many branches of mathematics as far as the overall idea of why the result is true is well transmitted, they are NOT acceptable in most safety critical computer applications.

For example, an incorrect proof may forget to treat a special case,

which is simple to handle but may need an additional non-invasive mathematical hypothesis, such as, for example, that three points must not be colinear. Unfortunately, such missed cases can lead to catastrophic errors in safety critical systems, where *all details are equally important*. For example, the three colinear points may generate a division-by-zero runtime error that may fail an entire application, regardless of how "inteligent" everything else is in it. A mechanical theorem prover will assist us during the proof, by keeping track of what is left to prove; also, it will only allow us to use a fixed number of inferenec rules, that have been apriory provided to the theorem prover and assumed as absolutely correct.

In other words, a theorem prover will not let us say the usual and often wrong "the rest of the proof is obvious".

Mathematical Proof of Associativity of Addition

We have to prove that the addition operation in the module

```
fmod NATURAL is
  sort Natural .
  op 0 : -> Natural [ctor] .
  op s : Natural -> Natural [ctor] .
  op _+_ : Natural Natural -> Natural .
  vars N M : Natural .
  eq N + 0 = N .
  eq N + s(M) = s(N + M) .
endfm
```

satisfies the *associativity* property,

$$(\forall \mathtt{N}, \mathtt{M}, \mathtt{L})\ \mathtt{N} + (\mathtt{M} + \mathtt{L}) = (\mathtt{N} + \mathtt{M}) + \mathtt{L}.$$

Mathematical Proof of Associativity of Addition (II)

We can prove the property by induction on `L`. That is, we prove it for `L` = `0` (base case) and then assuming that it holds for `L`, we prove it for `s(L)` (induction step).

Base Case: We need to show,

$$(\forall \texttt{N}, \texttt{M})\ \texttt{N + (M + 0)} = \texttt{(N + M) + 0}.$$

We can do this trivially, *by simplification* with the equation

```
eq N + 0 = N .
```

Mathematical Proof of Associativity of Addition (II)

Induction Step: We think of `L` as a *generic constant* (typically written n in textbooks) and assume that the associativity equation (*induction hypothesis* (IH))

$$(\forall \texttt{N}, \texttt{M})\ \texttt{N + (M + L)} = \texttt{(N + M) + L}.$$

holds for that constant. Then we try to prove the equation,

$$(\forall \texttt{N}, \texttt{M})\ \texttt{N + (M + s(L))} = \texttt{(N + M) + s(L)}.$$

using the induction hypothesis. Again, we can do this *by simplification* with the equations E in `NAT`, *and* the induction hypothesis IH equation, since we have,

Mathematical Proof of Associativity of Addition (III)

$$\texttt{N + (M + s(L))} \longrightarrow_E \texttt{N + s(M + L)}$$

$$\longrightarrow_E \texttt{s(N + (M + L))} \longrightarrow_{IH} \texttt{s((N + M) + L).}$$

and

$$\texttt{(N + M) + s(L)} \longrightarrow_E \texttt{s((N + M) + L).}$$

q.e.d

Machine-Assisted Proof with Maude's ITP

Maude's ITP is an *inductive theorem prover* supporting proof by induction in Maude functional modules. It is a program written entirely in Maude by Manuel Clavel in which one can:

- load in Maude the functional module or modules one wants to reason about
- load the file `itp-tool.maude` and then type `loop init .`
- enter *named goal* to be proved by the ITP *enclosed in parentheses* using the `goal` command.
- give *commands*, corresponding to proof steps, to prove that property, also *enclosed in parentheses*

Machine-Assisted Proof with Maude's ITP (II)

For example, suppose that we want to automatically prove the associativity of addition. We first load into Maude the module, say,

```
fmod NATURAL is
  sort Natural .
  op 0 : -> Natural [ctor] .
  op s : Natural -> Natural [ctor] .
  op _+_ : Natural Natural -> Natural .
  vars N M : Natural .
  eq N + 0 = N .
  eq N + s(M) = s(N + M) .
endfm
```

Then we load `itp-tool.maude` and type `loop init .`

Machine-Assisted Proof with Maude's ITP (III)

We then enter our associativity goal by giving it a name (`assoc`), mentioning the module in which it should be proved (`NATURAL`) and making explicit the *universal quantification* with the letter `A` and curly brackets notation. Note the required use of "on-the-fly" variables; and the generous use of parentheses to help the ITP parser.

```
(goal assoc : NATURAL |- A{N:Natural ; M:Natural ;  L:Natural}
   ((N:Natural + (M:Natural + L:Natural)) =
                    ((N:Natural + M:Natural) + L:Natural)) .)
```

The ITP then echoes, giving this goal an additional label ending (`$0`) to help the user keep track of where he/she is as the proof process unfolds and other (sub-)goals are generated.

```
==================================
assoc$0
==================================

|- A{N:Natural ; M:Natural ; L:Natural}
(N:Natural +(M:Natural + L:Natural)=
(N:Natural + M:Natural)+ L:Natural)

++++++++++++++++++++++++++++++++++

Maude>
```

Machine-Assisted Proof with Maude's ITP (IV)

We can then try prove goal `assoc$0` by induction on `L:Natural` by giving the command `(ind on L:Natural .)` and the tool then generates *two* subgoals (one for the base case, and another for the induction step; however, to avoid cluttering the output, it only displays the first of them:

```
==================================
assoc$1.0
==================================


|- A{N:Natural ; M:Natural}(N:Natural +(M:Natural + 0)=
                            (N:Natural + M:Natural)+ 0)

++++++++++++++++++++++++++++++++++
```

Machine-Assisted Proof with Maude's ITP (V)

We can then try prove the above "base case" subgoal by using the ITP's `auto` tactic that —after turning the variables into constants by the constants lemma (more on this later) and doing implication elimination if necessary— tries to simplify the goal by applying equations in the module, until hopefully reaching an identity. This tactic succeeds, leaving the second goal.

```
Maude> (auto .)
```

```
==================================
assoc$2.0
==================================

|- A{V0#0:Natural}
((A{N:Natural ; M:Natural}
(N:Natural +(M:Natural + V0#0:Natural)=
(N:Natural + M:Natural)+ V0#0:Natural))==>
(A{N:Natural ; M:Natural}
(N:Natural +(M:Natural + s(V0#0:Natural))=
(N:Natural + M:Natural)+ s(V0#0:Natural))))

++++++++++++++++++++++++++++++++++
```

Machine-Assisted Proof with Maude's ITP (VI)

We can likewise apply the `auto` tactic to the second goal, thus proving the associativity theorem.

```
Maude> (auto .)

q.e.d

++++++++++++++++++++++++++++++++++
```

Note that, in this case, both the constants lemma and implication elimination had to be invoked by `auto` before being able to simplify both sides of the conclusion using the induction hypothesis.

List Induction

So far, we have only used *natural number induction*. What about induction on other data structures? For example, what about *list induction*? Consider, for example, the following module defining a list `append` operator in terms of a list "cons" operator `_:_` for lists of natural numbers importing the `NAT` predefined module.

```
fmod LIST is protecting NAT .
sort List .
op nil : -> List [ctor] .
op _:_ : Nat List -> List [ctor] .
op append : List List -> List .
vars N M : Nat .
vars L L1 L2 L3 : List .
eq append(nil, L) = L .
eq append(N : L1, L2) = (N : append(L1, L2)) .
endfm
```

List Induction (II)

The `nil` constant and the "cons" operator `_:_` are *constructors* that play a role analogous to zero and successor in `NAT`, and list "append" is the analogous of number addition.

In fact, it is also *associative*, that is, the above module satisfies the property,

$(\forall \texttt{L1}, \texttt{L2}, \texttt{L3})\ \texttt{append(append(L1,L2),L3)} = \texttt{append(L1,append(L2,L3))}.$

List Induction (III)

The same scheme of proof used to prove associativity of addition can be used here as well, changing zero by `nil`, and successor by the "cons" operator `_:_`.

That is, if we want to do induction on `L1`, we must prove the base case for `nil`,

$(\forall \texttt{L2}, \texttt{L3})\ \texttt{append(append(nil,L2),L3)} = \texttt{append(nil,append(L2,L3))}.$

which follows trivially by simplification with the equation

```
eq append(nil, L) = L .
```

List Induction (IV)

And then we must prove the induction step by assuming that, considering `L1` as a *generic list constant*, we have the induction hypothesis equation,

$$(\forall \texttt{L2}, \texttt{L3})\ \texttt{append(append(L1,L2),L3)} = \texttt{append(L1,append(L2,L3))}.$$

that we try to use, along with the equations in the `LIST` module, to prove by simplification the equation

$$(\forall \texttt{L2}, \texttt{L3})\ \texttt{append(append((N : L1),L2),L3)} = \texttt{append((N : L1),append(L2,L3))}.$$

where `N` is a *generic natural constant*,

List Induction (V)

All this can again be done by hand, and it works. But it can be automated using the Maude ITP prover by:

- an induction step on `L`, which generates two subgoals, followed by
- `auto` steps for the subgoals (which succeed)

After initializing the ITP and entering the `LIST` module, we enter the main goal to the ITP. The screenshot shows the result of the `ind` step followed by the two `auto` steps, which complete the proof.

List Induction (VI)

```
Maude> (goal append-assoc :
    LIST |- A{L1:List ; L2:List ; L3:List}
                ((append(L1:List, append(L2:List, L3:List)))
                 = (append(append(L1:List, L2:List), L3:List))) .)

==================================
append-assoc$0
==================================

|- A{L1:List ; L2:List ; L3:List}
(append(L1:List,append(L2:List,L3:List))=
 append(append(L1:List,L2:List),L3:List))

++++++++++++++++++++++++++++++++++
```

```
Maude> (ind on L1:List .)

==================================
append-assoc$1.0
==================================

|- A{L2:List ; L3:List}
(append(nil,append(L2:List,L3:List))=
 append(append(nil,L2:List),L3:List))

++++++++++++++++++++++++++++++++++

Maude> (auto .)

==================================
append-assoc$2.0
==================================
```

```
|- A{V0#0:Nat ; V0#1:List}
((A{L2:List ; L3:List}
(append(V0#1:List,append(L2:List,L3:List))=
 append(append(V0#1:List,L2:List),L3:List)))
==>
(A{L2:List ; L3:List}
(append(V0#0:Nat : V0#1:List,append(L2:List,L3:List))=
 append(append(V0#0:Nat : V0#1:List,L2:List),L3:List))))

++++++++++++++++++++++++++++++++

Maude> (auto .)

q.e.d
```

Using Lemmas

Life is not always as easy as proving associativity of addition or of list append. Often, attempts at simplification using the `auto` tactic *do not succeed*. However, they *suggest lemmas to be proved*. Consider the following goal of proving *commutativity* of addition in our `NATURAL` module:

```
Maude> (goal comm : NATURAL |- A{N:Natural ; M:Natural}
         ((N:Natural + M:Natural) = (M:Natural + N:Natural)) .)

==================================
comm$0
==================================

|- A{N:Natural ; M:Natural}
(N:Natural + M:Natural = M:Natural + N:Natural)
```

Using Lemmas (II)

We can try to prove it by induction on `M:Nat`

```
Maude> (ind on M:Natural .)

==================================
comm$1.0
==================================


|- A{N:Natural}(N:Natural + 0 = 0 + N:Natural)
```

Using Lemmas (III)

When we apply the `auto` tactic to this first goal we get,

```
Maude> (auto .)

==================================
comm$1.1.1.1.0
==================================

|- N*Natural = 0 + N*Natural
```

Using Lemmas (IV)

What we can do is to *assume the unsimplified equation* yielded by `auto` *as a lemma* in the proof of our main goal. We can do this by giving this lemma a label and adding it to the module of goal `comm$1.1.1.1.0` as follows:

```
Maude> (lem 0-comm : A{N:Natural}((0 + N:Natural) = (N:Natural)) .)

==================================
0-comm$0
==================================

|- A{N:Natural}(0 + N:Natural = N:Natural)
```

Using Lemmas (V)

Adding this lemma creates a new goal `0-comm$0`, that is, a new proof obligation that we need to discharge. We can do so by proving the lemma by induction on `N:Natural`, using the `auto` tactic to eliminate the two generated subgoals:

```
Maude> (ind on N:Natural .)

==================================
0-comm$1.0
==================================

|- 0 + 0 = 0

++++++++++++++++++++++++++++++++++
```

```
Maude> (auto .)
==================================
0-comm$2.0
==================================

|- A{V1#0:Natural}
((0 + V1#0:Natural = V1#0:Natural)==>
 (0 + s(V1#0:Natural)= s(V1#0:Natural)))

++++++++++++++++++++++++++++++++++

Maude> (auto .)
==================================
comm$1.1.1.1.0
==================================

|- N*Natural = 0 + N*Natural
```

Using Lemmas (VI)

Proving now our first original subgoal becomes automatic (because of the lemma) but we are then faced with the second original subgoal:

```
==================================
comm$1.1.1.1.0
==================================

|- N*Natural = 0 + N*Natural

++++++++++++++++++++++++++++++++++

Maude> (auto .)

==================================
comm$2.0
```

```
==================================

|- A{V0#0:Natural}
((A{N:Natural}
 (N:Natural + V0#0:Natural = V0#0:Natural + N:Natural))
==>
(A{N:Natural}
 (N:Natural + s(V0#0:Natural)= s(V0#0:Natural)+ N:Natural)))
```

Using Lemmas (VII)

We can apply also the `auto` tactic to the remaining goal `comm$2.0`, but, again, we get an unproved equality that we can use as a suggestion for a new lemma.

```
Maude> (auto .)

==================================
comm$2.1.1.1.1.1.0
==================================

|- s(V0#0*Natural + N*Natural)= s(V0#0*Natural)+ N*Natural
```

Using Lemmas (IX)

We can again enter and prove this lemma by induction on `N:Natural` and two applications of the `auto` tactic, which brings us back to our last unproved subgoal.

```
Maude> (ind on N:Natural .)
==================================
s-comm$1.0
==================================

|- A{M:Natural}(s(M:Natural + 0)= s(M:Natural)+ 0)

++++++++++++++++++++++++++++++++++

Maude> (auto .)
==================================
s-comm$2.0
```

```
==================================

|- A{V1#0:Natural}
((A{M:Natural}
(s(M:Natural + V1#0:Natural)= s(M:Natural)+ V1#0:Natural))
==>
(A{M:Natural}
(s(M:Natural + s(V1#0:Natural))= s(M:Natural)+ s(V1#0:Natural))))

++++++++++++++++++++++++++++++++++

Maude> (auto .)
==================================
comm$2.1.1.1.1.1.0
==================================

|- s(V0#0*Natural + N*Natural)= s(V0#0*Natural)+ N*Natural
```

Using Lemmas (X)

Finally, since we have assumed our just-proved second lemma, we can now eliminate this last subgoal automatically:

```
==================================
comm$2.1.1.1.1.1.0
==================================

|- s(V0#0*Natural + N*Natural)= s(V0#0*Natural)+ N*Natural

++++++++++++++++++++++++++++++++++

Maude> (auto .)

q.e.d
```

Caveats on the ITP Tool

The ITP tool is for the moment an *experimental system*, with limited support for *error messages*. Therefore, if you run into parsing troubles entering a goal or a command, besides consuting the ITP Manual to make sure you did things right, you may also *use parentheses generously* in all goals, lemmas, and other ITP commands to help the ITP parser.

Readings and Exercises

Study the description of ITP commads in the ITP manual, which can be found in the course web page.

Look at, and play with, some examples of ITP proofs, which are stored, together with the files for the ITP in the course web page.

Try to prove: (1) associativity and commutativity of natural number multiplication, and (2) the list equation `rev(rev(L)) = L`, for your favorite specifications of mutiplication, and of the `rev` function that reverses a list, using the ITP tool.

The ITP Inference Rules

Notice that in the ITP we *reason backwards*, replacing the *main goal* G we want to prove by *subgoals*, $G_1, \ldots, G_n$, such that if we prove each of the subgoals, then we have proved the main goal.

For such an inference to be *sound*, the implication

$$G_1 \wedge \ldots \wedge G_n \Rightarrow G$$

should always be *satisfied*, that is, should be *semantically valid* in the initial model $T_{\Sigma,E}$ on which we are doing the inductive reasoning.

The ITP Inference Rules (II)

Such semantically valid inferences are expressed as inference rules

$$\frac{G_1 \quad \ldots \quad G_n}{G}$$

However, since we are reasoning *backwards*, from the root of the proof tree to the leaves, the ITP uses such rules in the *opposite direction*, as rules

$$\frac{G}{G_1 \quad \ldots \quad G_n}$$

We will illustrate through an example such backward reasoning for several ITP inference rules besides the induction rule, and will at the same time *justify their soundness*.

Linearity of the Number Ordering

Consider the following module defining the order on numbers, which we would like to prove is linear.

```
(fmod NATURAL-ORD is
  sort Natural .
  op 0 : -> Natural [ctor] .
  op s : Natural -> Natural [ctor] .
  op _<_ : Natural Natural -> Bool .
  op _=<_ : Natural Natural -> Bool .
  vars N M : Natural .
  eq N < 0 = false .
  eq 0 < s(N) = true .
  eq s(N) < s(M) = N < M .
  ceq N =< M = true if N < M .
  ceq N =< M = true if not(M < N) .
  ceq N =< M = false if M < N .
endfm)
```

The cns Inference Rule

After entering the goal stating that the order on the naturals is linear, one of the possible ITP inference rules we can invoke is the *lemma of constants*, which converts universally quantified variables in a goal into constants.

```
Maude> (goal linear : NATURAL-ORD |- {N:Natural ; M:Natural}
       (((N:Natural =< M:Natural) or
        (M:Natural =< N:Natural)) = true) .)

==================================
linear$0
==================================


|-{N:Natural ; M:Natural}(N:Natural =< M:Natural or
                          M:Natural =< N:Natural = true)
```

```
++++++++++++++++++++++++++++++++++

Maude> (cns linear$0 .)

==================================
linear$0
==================================


|- N*Natural =< M*Natural or M*Natural =< N*Natural = true
```

The cns Inference Rule (II)

The fact that this is a semantically valid inference is based on the **Constants Lemma**, which states the equivalence between satisfiability of a quantified equation, and of the same equation with the variables transformed into *generic constants*,

$$E \models_{\Sigma} (\forall X)\ t = t' \quad \Leftrightarrow \quad E \models_{\Sigma(X)} (\forall \emptyset)\ t = t'.$$

Thanks to the *completeness* of equational reasoning, this is expressed in the ITP as the ***cns*** rule,

$$\frac{E \vdash_{\Sigma} (\forall X)\ t = t'}{E \vdash_{\Sigma(X)} (\forall \emptyset)\ t = t'}$$

Optional: Justification of the cns Rule

We can *justify* the validity of the **Constants Lemma** not only for unconditional equations, but also for conditional ones as follows. Given a conditional equation

$$(\forall X)\ \varphi = (\forall X)\ t = t' \quad \Leftarrow \quad u_1 = v_1 \wedge \ldots \wedge u_n = v_n,$$

we have $E \models_\Sigma (\forall X)\ \varphi$ iff, by definition,

$$(\forall (A,a) \in \mathbf{Alg}_{\Sigma(X)})\ A \models_\Sigma E \quad \Rightarrow \quad (A,a) \models_{\Sigma(X)} (\forall \emptyset)\ \varphi,$$

which is equivalent, by **Ex**6.2, to

$$(\forall (A,a) \in \mathbf{Alg}_{\Sigma(X)})\ (A,a) \models_{\Sigma(X)} E \quad \Rightarrow \quad (A,a) \models_{\Sigma(X)} (\forall \emptyset)\ \varphi,$$

which, by definition of satisfaction, is precisely $E \models_{\Sigma(X)} (\forall \emptyset)\ \varphi$, as desired.

Reasoning by Cases: The `split` Rule

To prove our goal, we can now *reason by cases*. For each pair of natural numbers n, m, either $n < m = \texttt{true}$, or $n < m = \texttt{false}$. Therefore, we can *split* a goal involving n and m into two subgoals: one in which we *assume* $n < m = \texttt{true}$ as an extra hypothesis, and another in which we *assume* $n < m = \texttt{false}$.

In the ITP this is accomplished by the `split` rule, which, given an *unquantified goal without variables* (only "generic" constants like n, m) and given a Boolean-valued expression involving some of those generic constants, splits a given goal into two: one assuming the expression `true`, and another assuming it `false`.

Reasoning by Cases: The `split` Rule (II)

In our example we can give the `split` command,

```
Maude> (split linear$0 on (N*Natural < M*Natural) . )

==================================
linear$1.0
==================================


|- N*Natural =< M*Natural or M*Natural =< N*Natural = true
==================================
linear$2.0
==================================


|- N*Natural =< M*Natural or M*Natural =< N*Natural = true
```

Reasoning by Cases: The `split` Rule (III)

The goals remain the same, but the split hypotheses for each case have been added to each case's module. Using these hypotheses we can now discharge each of the subgoals with the `auto` tactic.

```
Maude> (auto linear$1.0 .)
==================================
linear$2.0
==================================

|- N*Natural =< M*Natural or M*Natural =< N*Natural = true

++++++++++++++++++++++++++++++++++

Maude> (auto linear$2.0 .)
q.e.d
```

Caveats on the `split` Rule

As already mentioned, the present version of the ITP requires that the goal to which the `split` rule is applied is an *unquantified goal without variables*, in which only "generic" constants —such as `N*Natural` and `M*Natural` in our example— appear.

This requirement will be relaxed in future ITP versions, but it is assumed and required by the present version. Therefore, applications of `split` to *quantified goals with variables* are currently *forbidden*: in the present ITP version we must first *transform* such variables into generic constants using `cns`.

Applications of `split` Must Protect `BOOL`

Regardless of the current ITP restrictions in the application of `split`, there is a fundamental way in which the application of `split` would be unsound, namely, if we have *messed up* the Booleans by adding *junk* and perhaps *confusion* to them, so that we do not have anymore two different canonical forms, `true`, and `false`, but may have other nonstandard Boolean elements as well.

This can happen because, when defining new predicates with operators of sort `Bool`, we *do not give enough equations*, thus adding "junk" to `Bool`, or we give *the wrong equations*, adding "confusion" and possibly also "junk" to `Bool`. Therefore, sound application of `split` require that the `BOOL` submodule is *protected*.

Optional: Protecting Module Importations

In general, if we have a theory (Σ, E) having a subtheory (Σ', E') with,

$$\Sigma' \subseteq \Sigma \quad \text{and} \quad E' \subseteq E,$$

and the module `fmod`(Σ, E)`endfm` imports the submodule `fmod`(Σ', E')`endfm` in *`protecting`* mode, we require that the unique Σ'-homomorphism

$$eval^{E'}_{T_{\Sigma,E}|_{\Sigma'}} : T_{\Sigma'/E'} \longrightarrow T_{\Sigma,E}|_{\Sigma'}$$

is an *isomorphism*.

Optional: Justification of the `split` Rule

Suppose that we are reasoning inductively about a module $\texttt{fmod}(\Sigma, E)\texttt{endfm}$, which correctly imports `BOOL` in *protecting* mode (**Note**: this must be checked independently, as an *implicit proof obligation*).

In `BOOL` we have,

$$T_{\texttt{BOOL}} \models \texttt{true} \neq \texttt{false}$$

and we also have,

$$T_{\texttt{BOOL}} \models (\forall x : \texttt{Bool})\ x = \texttt{true} \vee x = \texttt{false}$$

Notice that, by the *protecting* importation, we have, $T_{\Sigma/E}|_{\Sigma_{\texttt{BOOL}}} \cong T_{\texttt{BOOL}}$.

Optional: Justification of the `split` Rule (II)

Therefore, given any Boolean valued Σ-term $p \in T_\Sigma(X)_{\texttt{Bool}}$, and given any assignment $a : X \longrightarrow T_{\Sigma/E}$, we have,

$$(T_{\Sigma/E}, a) \not\models (p \neq \texttt{true} \wedge p \neq \texttt{false}).$$

That is, for any such assignment a, the above proposition is *equivalent* to the *identically false* proposition, $\bot$, which is never satisfied:

$$(T_{\Sigma/E}, a) \models (p \neq \texttt{true} \wedge p \neq \texttt{false}) \quad \Leftrightarrow \quad (T_{\Sigma/E}, a) \models \bot.$$

Now, since for any proposition A we always have the Boolean equivalence, $A \equiv A \vee \bot$, we also have an equivalence $(\forall X)\ A \equiv (\forall X)\ (A \vee \bot)$. Therefore, given an equation $(\forall X)\ t = t'$, we have,

Optional: Justification of the `split` Rule (III)

$$T_{\Sigma/E} \models (\forall X)\, t = t' \;\; \Leftrightarrow \;\; T_{\Sigma/E} \models (\forall X)\, ((t = t') \vee \bot)$$

or, equivalently,

$$T_{\Sigma/E} \models (\forall X)\, t = t' \;\; \Leftrightarrow \;\; T_{\Sigma/E} \models (\forall X)\, ((t = t') \vee (p \neq \texttt{true} \wedge p \neq \texttt{false}))$$

which, using distributivity of disjunction over conjunction, plus the fact that $A \Rightarrow B \equiv (\neg A) \vee B$, is equivalent to,

$$T_{\Sigma/E} \models (\forall X)\, t = t' \;\; \Leftrightarrow$$
$$T_{\Sigma/E} \models (\forall X)\, (p = \texttt{true} \Rightarrow t = t') \wedge (p = \texttt{false} \Rightarrow t = t'),$$

which, modulo the distribution of $\forall$ over $\wedge$, is our desired justification for `split` as a sound inductive reasoning rule.

Induction on Other Data Structures: Tree Induction

We have already seen examples of how the ITP's `ind` rule applies to natural number induction and to list induction.

Before discussing the *most general* form of the `ind` rule for any signature of constructors Ω and its justification in the next lecture, we give an example illustrating *binary tree induction*, in which the data in leaves are seen as depth-zero trees.

The intuitive idea is that to prove an inductive property P about such trees we must show: (1) that P holds for the data elements (**base case**); and (2) that if P holds for the left and right subtrees, then it must hold for their binary join (**induction step**).

Induction on Other Data Structures: Tree Induction (II)

Consider the following module defining binary trees whose nodes are quoted identifiers (constants in the predefined module `QID`), and a reverse function on binary trees.

```
(fmod TREE is
protecting QID .
sort Tree .
subsort Qid < Tree .
op _#_ : Tree Tree -> Tree [ctor] .
op rev : Tree -> Tree .
var I : Qid .
vars T T' : Tree .
eq rev(I) = I .
eq rev(T # T') = rev(T') # rev(T) .
endfm)
```

Induction on Other Data Structures: Tree Induction (III)

We can apply binary tree induction to prove that for all trees `T` the equation `rev(rev(T)) = T` holds. We can do so by entering the `TREE` module in the ITP and the goal:

```
Maude> (goal rev : TREE |- {T:Tree}((rev(rev(T:Tree))) = (T:Tree)) .)

==================================
rev$0
==================================


|-{T:Tree}(rev(rev(T:Tree))= T:Tree)
```

Induction on Other Data Structures: Tree Induction (IV)

We can then try to prove this goal by induction on the variable `T:Tree`.

```
Maude> (ind rev$0 on T:Tree .)

==================================
rev$1.0
==================================


|-{V2#0:Tree ; V2#1:Tree}(rev(rev(V2#1:Tree))= V2#1:Tree & rev(rev(V2#0:Tree))=
    V2#0:Tree ==> rev(rev(V2#0:Tree # V2#1:Tree))= V2#0:Tree # V2#1:Tree)
==================================
rev$2.0
==================================


|-{V2#0:Qid}(rev(rev(V2#0:Qid))= V2#0:Qid)
```

Induction on Other Data Structures: Tree Induction (V)

Note that goal `rev$1.0` is the "induction step" in tree induction, whereas the "base case" is goal `rev$2.0`. Both subgoals can then be proved using the `auto` tactic.

```
Maude> (auto rev$1.0 .)

==================================
rev$2.0
==================================


|-{V2#0:Qid}(rev(rev(V2#0:Qid))= V2#0:Qid)

++++++++++++++++++++++++++++++++++

Maude> (auto rev$2.0 .)

q.e.d
```

Structural Induction

We have already observed how the ITP supports inductive proofs in three cases: natural number induction, list induction, and tree induction. But what is the *general* form of induction supported by the ITP for a specification having a subsignature Ω of constructors? This general form is called *structural induction*. It reduces proving an inductive property of the form $(\forall x : s)\ P(x)$, to proving:

- **Base Case**. For any constant $a : nil \longrightarrow s'$ in Ω with $s' \leq s$, the subgoal $P(x/a)$

  Notation: Given a variable x, the substitution mapping x to a term t is denoted (x/t), and its homomorphic extension $\overline{(x/t)}$ is also denoted (x/t).

Structural Induction (II)

- **Induction Step**. For each constructor $f : s_1 \ldots s_n \longrightarrow s'$ in Ω with $s' \leq s$, where the sorts $s_{i_1}, \ldots, s_{i_k}$ are those among the $s_1 \ldots s_n$ such that $s_{i_j} \leq s$, $1 \leq j \leq k$, the subgoal,

  $$(\forall x_1 : s_1, \ldots, x_n : s_n)\ P(x/x_{i_1}) \wedge \ldots \wedge P(x/x_{i_k}) \Rightarrow P(x/f(x_1, \ldots, x_n)).$$

  Note: It may happen that *none* of the sorts among the $s_1 \ldots s_n$ is s or a subsort of s. In that case, the subgoal takes the form $(\forall x_1 : s_1, \ldots, x_n : s_n)\ P(x/f(x_1, \ldots, x_n))$.

- **Subsorts Without Constructors**. If $s' \leq s$ is a subsort having no constructor constants or operators in a sort $s'' \leq s'$, then we add the subgoal $(\forall y : s')\ P(x/y)$.

Structural Induction (III)

Note: If the signature Ω of constructors has been fully specified, the base case and the induction step *implicitly cover all subsorts*, so that the third case should never arise, except perhaps for s' an *empty* sort, with no terms whatsoever, for which the property P then trivially holds.

Therefore, from now on we will *systematically ignore* the case of subsorts without constructors in the rest of our theoretical discussions. In practice, however, this case is actually quite useful, and this for two reasons:

Structural Induction (IV)

- to deal with *constants in predeclared modules*, such as `QID`, which are built-in and are not defined as constructors (we encountered this phenomenon in our tree-reverse example); and

- to generalize these inductive proof methods to *parameterized modules*, such as `LIST(X)`, where the parametric sort of elements might be a subsort of the sort `List(X)`, but we have no a priori information about the constructors of such a parametric sort of elements.

Structural Induction (V)

Ignoring the case of subsorts without constructors, this then becomes an inductive inference rule of the form,

$$\frac{\bigwedge_i P(x/a_i) \wedge \bigwedge_j (\forall \overline{x} : \overline{s})\ P(x/x_{i_1}) \wedge \ldots \wedge P(x/x_{i_k}) \Rightarrow P(x/f_j(x_1, \ldots, x_{n_j}))}{(\forall x : s)\ P(x)}$$

where the a_i and the f_j include all the constructor constants and operators meeting the properties specified above, and where $(\forall \overline{x} : \overline{s})$ abbreviates $(\forall x_1 : s_1, \ldots, x_{n_j} : s_{n_j})$.

Of course, in the ITP this rule is used backwards as the `ind` rule,

$$\frac{(\forall x : s)\ P(x)}{\bigwedge_i P(x/a_i) \wedge \bigwedge_j (\forall \overline{x} : \overline{s})\ P(x/x_{i_1}) \wedge \ldots \wedge P(x/x_{i_k}) \Rightarrow P(x/f_j(x_1, \ldots, x_{n_j}))}$$

Optional: Justification of the `ind` Rule

Why is `ind` a *sound* inference rule? First consider:

Lemma: Given a confluent and terminating equational theory (Σ, E) with subsignature of constructors Ω, and given any Σ-equation $(\forall X)\ t = t'$, we have

$$T_{\Sigma/E} \models (\forall X)\ t = t' \quad \Leftrightarrow \quad (\forall \theta : X \longrightarrow T_\Omega)\ T_{\Sigma/E} \models (\forall \emptyset)\ \overline{\theta} t = \overline{\theta} t'.$$

Proof: Notice that, since Ω is the subsignature of constructors, any assignment $a : X \longrightarrow T_{\Sigma/E}$ factors as $a = \theta; eval_{T_{\Sigma/E}|\Omega}$, where if $a(x) = [t]$, we can define $\theta : X \longrightarrow T_\Omega$ by $\theta(x) = can_E(t)$. Therefore, by the usual factorization and initiality argument,

$$(T_{\Sigma/E}, a) \models t = t' \quad \Leftrightarrow \quad T_{\Sigma/E} \models (\forall \emptyset)\ \overline{\theta} t = \overline{\theta} t'. \quad \text{q.e.d.}$$

Optional: Justification of the `ind` Rule (II)

Notice that the argument of the above lemma does not depend on our formula being actually an equation: it would similarly apply to conditional equations, and even to general, universally-quantified first-order formulas.

Therefore, we have reduced the problem of proving an inductive property, $(\forall x : s)\ P(x)$, to that of proving that for all $t \in T_{\Omega,s}$ the instantiated property $P(x/t)$ holds.

Here is where structural induction steps in as a method, namely, by analyzing more closely what it means to prove something for all $t \in T_{\Omega,s}$.

Optional: Justification of the `ind` Rule (III)

By the very inductive definition of $T_{\Omega,s}$, a term t is in $T_{\Omega,s}$ iff either:

1. $t = a$, for $a : nil \longrightarrow s'$ a constant in Ω with $s' \leq s$; or
2. there is a constructor $f : s_1 \ldots s_n \longrightarrow s'$ in Ω with $s' \leq s$ (where the sorts $s_{i_1}, \ldots, s_{i_k}$ are those among the $s_1 \ldots s_n$ such that $s_{i_j} \leq s$, $1 \leq j \leq k$) and terms $t_i \in T_{\Omega,s_i}$, $1 \leq i \leq n$ such that $t = f(t_1, \ldots, t_n)$

Therefore, given a property $P(x)$ with x a variable of sort s, if we prove that:

Optional: Justification of the `ind` Rule (IV)

1. for each constant a as above, $P(x/a)$ holds; and
2. for each f as above, assuming that $P(x/t_{i_1}), \ldots, P(x/t_{i_k})$, holds then we prove that $P(x/f(t_1, \ldots, t_n))$ holds,

   (where $t_{i_j} \in T_{\Omega,s_{i_j}}$, and s_{i_j}, $1 \leq j \leq k$ are those sorts among the $s_1 \ldots s_n$ for the arguments of f such that $s_{i_j} \leq s$, $1 \leq j \leq k$)

then we have proved that $P(x/t)$ holds *for all* $t \in T_{\Omega,s}$ (indeed, 1–2 above amount to a proof by induction on the *depth* of $t \in T_{\Omega,s}$) which, by our previous lemma, shows,

$$T_{\Sigma/E} \models (\forall x : s)\ P(x),$$

and therefore justifies `ind` as a sound inference rule.

Optional: Need to Check Sufficient Completeness

All this is fine, but there is a pending issue. How do we *know* that the declared subsignature of constructors is correct? We need to check that it is sufficiently complete, for example using the SCC tool.

For example, as discussed in the justification of the `split` rule, the user may have overlooked giving *enough equations* for the defined functions, and then it becomes impossible to simplify every ground term to a constructor term.

A quick glance at our proof of the lemma involved in justifying the soundness of the `ind` rule shows that the reduction of proving $P(x)$ to proving $P(x/t)$ for each $t \in T_{\Omega,s}$ works just the same under *much weaker assumptions* than (Σ, E) confluent, sort-decreasing, and terminating with subsignature of constructors Ω.

Verification of Functional Modules

We are now ready to consider a *general methodology* for verifying *declarative programs*. We will present the ideas in the context of verifying Maude *functional modules*, which are based on equational logic. The first key observation is that there are *three viewpoints* involved:

- the *customer's viewpoint*, expressed in the form of *requirements* that the desired software should satisfy;
- the *implementor's viewpoint*, whose job is to write a program meeting the customer's requirements; and
- the *verifier's viewpoint*, whose responsibility is to verify that the implementation does indeed meet the customer's requirements.

The Customer's Requirements and Specification

The customer's *requirements* may generally be *informal*.
Furthermore, they may involve *other concerns beyond correctness*, such as user-friendliness, a good graphical user interface, performance requirements, requirements about the underlying hardware and systems software, interoperability requirements, and so on.

Program verification focuses primarily on *correctness requirements*, which are always important, but may be crucial for safety-critical applications, were incorrect software may cause loss of human lives and/or other important damages.

The Customer's Requirements and Specification (II)

To make possible the high assurance of correctness afforded by *mathematical verification*, such correctness requirements must be *formalized*, typically in the form of a *logical theory* T_{spec}, stating precisely the customer's (correctness) *specification*.

This capture of the informal correctness requirements into a formal specification can be done by the customer himself, or by an expert aiding the customer in this task. It is of course very important to make sure that the formal specification *captures faithfully* the informal requirements.

The Customer's Formal Specification

In the context of Maude functional modules, it is reasonable to assume that such a formal specification will take the form of a theory,

$$T_{spec} = (\Delta, E_0 \cup Q)$$

where:

- (Δ_0, E_0), with $\Delta_0 \subseteq \Delta$, is an equational theory, that could be called the *framework theory*, specifying things such as key data structures and functions, including auxiliary functions needed to state key properties, and
- Q is a collection of sentences in first-order logic, specifying the actual *correctness properties* that the software must satisfy within the (Δ, E_0) framework.

Customer Specification: A Sorting Example

We can illustrate these ideas with a simple example, namely a customer who wants a sorting program to sort lists of integers.

As already mentioned, the customer's requirements may involve other important considerations, such as reasonable efficiency; for example, that it returns answers in time at most quadratic on the size of the input list.

Informally, the correctness requirement seems both obvious and tautological, namely, *the program should return the input list in sorted form*.

Customer Specification: A Sorting Example (II)

However, in order to *prove* that a given implementation satisfies such a requirement, we need to *capture* such an informal requirement in a formalized way *as a theory* T_{spec}.

We must specify two things:

1. the *data*, namely lists of integers, and some auxiliary functions, and
2. the sorting function and its *properties*.

Customer Specification: A Sorting Example (III)

Specifying the properties of the sorting function is not entirely trivial:

- first of all, we need to make precise what we mean by a list being *sorted*;
- but it is not enough to just require that the result is sorted: a function returning always the empty list will satisfy such a requirement! The original list and the sorted list should have *the same elements*.

All this can be stated precisely in three equational theories:

Customer Specification: A Sorting Example (IV)

First, the *data*, say lists of numbers, is specified in a module such as the following `INT-LIST` module

```
fmod INT-LIST is protecting INT .
  sorts List .
  op nil : -> List [ctor] .
  op _:_ : Int List -> List [ctor] .
endfm
```

Customer Specification: A Sorting Example (V)

Then, a *framework theory* importing INT-LIST,

```
fmod FRAME-SORTING-REQUIREMENTS is protecting INT-LIST .
  sort Multiset .
  subsort Int < Multiset .
  op sorted : List -> Bool .
  op null : -> Multiset .
  op _ _ : Multiset Multiset -> Multiset [assoc comm id: null] .
  op mset : List -> Multiset .
  vars N M : Int .
  var L : List .
  eq sorted(nil) = true .
  eq sorted(N : nil) = true .
  ceq sorted(N : M : L) = sorted(M : L)  if (N <= M) = true .
  ceq sorted(N : M : L) = false  if N <= M = false .
  eq mset(nil) = null .
  eq mset(N : L) = N mset(L) .
endfm
```

Customer Specification: A Sorting Example (VI)

Finally, a functional *theory* specifying two key requirements for the `sort` function:

```
(fth SORTING-REQUIREMENTS is
  protecting FRAME-SORTING-REQUIREMENTS .
  op sort : List -> List .
  var L : List .
  eq sorted(sort(L)) = true .
  eq mset(sort(L)) = mset(L) .
endfth)
```

Customer Specification: A Sorting Example (VII)

This is an instance of our general methodology, where the correctness specification has the form, $T_{spec} = (\Delta, E_0 \cup Q)$. Here, the framework theory (Δ_0, E_0) is the module `FRAME-SORTING-REQUIREMENTS`, and the theory T_{spec} itself is `SORTING-REQUIREMENTS`.

Note that the theory T_{spec} is a Full Maude specification that has been introduced with the keywords `fth`T_{spec}`endth`. This means that it has a *loose semantics*; that is, we do not require its models to be initial. However, because of the keyword `protecting` `FRAME-SORTING-REQUIREMENTS`, the functional submodule `FRAME-SORTING-REQUIREMENTS` is imported *with its initial semantics*.

Customer Specification: A Sorting Example (VIII)

Mathematically, what this means is that, for A to be an acceptable model of T_{spec}, besides having to satisfy the axioms in T_{spec}, the data types of lists, multisets, integers, and of booleans, as well as all the functions defined on them by initial algebra semantics (with keywords `fmodFRAME-SORTING-REQUIREMENTSendfm`) must be respected. In short, we must have an *isomorphism*,

$$A|_{\Sigma_{\texttt{FRAME-SORTING-REQUIREMENTS}}} \cong T_{\texttt{FRAME-SORTING-REQUIREMENTS}}.$$

The Implementation: Insert-Sort

One possible Maude implementation is *insert-sort*,

```
fmod INSERT-SORT is
  protecting INT-LIST .
  op ins : Int List -> List .
  op sort : List -> List .
  vars N M : Int .
  var L : List .
  eq ins(N, nil) = N : nil .
  ceq ins(N, M : L) = N : M : L if N <= M = true .
  ceq ins(N, M : L) = M : ins(N, L) if N <= M = false .
  eq sort(nil) = nil .
  eq sort(N : L) = ins(N, sort(L)) .
endfm
```

The Implementation: Insert-Sort (II)

The module T_{imp} in our general methodology, becomes in this case the Maude module `INSERT-SORT`.

Note that the auxiliary function `ins` defined in `INSERT-SORT` *for implementation purposes* is *completely different* from the auxiliary functions, `sorted`, `_ _`, and `mset` defined in `FRAME-SORTING-REQUIREMENTS` for *specification purposes*, to formally capture the customer's correctness requirements.

Therefore, the *signatures* of T_{spec} and T_{imp} *do not necessarily coincide*. However, *for verification purposes*, they are both *included as subsignatures* in T_{ver}.

The Theory T_{ver}

For verification purposes we typically *need to use the auxiliary functions* defined in *both* the framework theory (Δ_0, E_0) and in $T_{imp} = (\Sigma, E)$. This means that the theory $T_{ver} = (\Sigma', E')$ will typically have theory inclusions,

- ($\flat$) $(\Delta_0, E_0) \subseteq (\Sigma', E')$, and
- ($\dagger$) $(\Sigma, E) \subseteq (\Sigma', E')$.

The main goal of the verification effort is then to establish,

$$T_{ver} \vdash_{ind} Q.$$

But for this inductive inference to be applicable to the implementation theory $T_{imp} = (\Sigma, E)$, we need to require that ($\dagger$) is a *protecting inclusion*, so that we have an isomorphism, $T_{\Sigma'/E'}|_{\Sigma} \cong T_{\Sigma/E}$.

The Theory T_{ver} and its Verification

In this example, the theory T_{ver} must contain *both* the framework theory and the implementation theory:

```
fmod INSERT-SORT-VERIFICATION is
protecting INSERT-SORT .
protecting FRAME-SORTING-REQUIREMENTS .
endfm
```

We can then give this theory to the ITP and prove in it as theorems the two equations in T_{spec}, namely,

```
eq sorted(sort(L)) = true .
eq mset(sort(L)) = mset(L) .
```